To appear in the proceedings of *Formal Methods and Models for Codesign* (MEMOCODE), 2007.

Temporal Refinement Using SMT and Model Checking with an Application to Physical-Layer Protocols

Geoffrey M. Brown
Indiana University, Bloomington
geobrown@cs.indiana.edu

Lee Pike
Galois, Inc.
leepike@galois.com

Abstract

This paper demonstrates how to use a satisfiability modulo theories *(SMT) solver together with a bounded model checker to prove temporal refinement conditions. The method is demonstrated by refining a specification of the 8N1 protocol, a widely-used protocol for serial data transmission. A nondeterministic finite-state 8N1 specification is refined to an infinite-state implementation in which interleavings are constrained by real-time linear inequalities. The refinement proof is via automated induction proofs over infinite-state transitions systems using SMT and model checking, as implemented in SRI International's* Symbolic Analysis Laboratory *(SAL).*

1 Introduction

A recently-developed formal verification technique combines a *satisfiability modulo theories* (SMT) solver and a bounded model checker to prove LTL safety properties over infinite-state transition systems. The proof technique implemented is k-*induction*, a generalization of induction over (infinite-state) transition systems; for brevity, we will call the verification technique *infinite-bmc induction* (for *infinite*-state *b*ounded *m*odel *c*hecking *induction*) [4, 13]. One implementation of infinite-bmc induction is in SRI International's *Symbolic Analysis Laboratory* (SAL) [3]. In this paper, we apply infinite-bmc induction to easily prove the correctness of a class of real-time protocols.

Traditionally in model checking, a finite-state abstraction of a real-time protocol is used to model the passage of time with an asynchronous interleaving semantics. The benefit of such an abstraction is that if the finite-state model is not too large, it can be verified automatically using standard model checking techniques (e.g., BDDs). Furthermore, a finite-state model of the protocol can be composed with a finite-state model of synchronous hardware, and the composition may also be model-checked. However, to ensure the that safety properties proved of the finite-state model hold of the infinite-state one, in which the progress of time is modeled with better fidelity, one must complete a refinement proof from the former to the latter.

In this paper, we describe a simple refinement approach, derived from the classic Abadi-Lamport refinement method [1]. We emphasize that the approach described is one to refine *temporal* constraints and does not address other refinements such as data refinement. We demonstrate how a specification can be formally refined into a more deterministic implementation. The refinement ensures that the safety properties that hold of the finite-state model also hold of the infinite-state model. As a case-study, we demonstrate the approach by refining a finite-state model of the 8N1 protocol, a common physical-layer protocol used for serial data transmission between independently-clocked hardware (described in detail in Section 2.2), into an implementation that explicitly captures real-time constraints by linear inequalities over the real numbers. To prove the correctness of the protocol, we use BDDs. To prove the refinement holds, we use infinite-bmc induction. Besides a method of temporal refinement, this paper introduces a succinct and general constraint-based model of physical-layer protocols that simplifies the proofs and decomposes the model of the environment (i.e., the effects of metastability) and the protocol specification. Specifications and proofs are also available for refinements of the Biphase Mark protocol and a version of the 8N1 protocol that includes shift-registers.[1]

The remainder of the paper is organized as follows. In Section 2, we describe related work, the 8N1 protocol, and the SAL model checker. The finite-state specification of the 8N1 protocol and its proof of correctness is presented in Section 3. The infinite-state implementation of 8N1 is pre-

[1]The specifications and proofs in SAL associated with this paper are available at `http://www.cs.indiana.edu/~lepike/pub_pages/refinement.html`. SAL can be obtained at `http://fm.csl.sri.com/`.

sented in Section 4. We verify the refinement of the specification to the implementation in Section 5. We conclude with a brief discussion in Section 6.

2 Related Work and Background

In Section 2.1 we highlight previous work in physical-layer protocol verification. In Section 2.2, we describe the 8N1 protocol, and in Section 2.3, we describe the SAL model checker.

2.1 Physical-Layer Protocol Verification

The first infinite-bmc induction verification of a real-time protocol was Dutertre and Sorea's verification of the startup protocol for the Time-Triggered Architecture, a fault-tolerant bus [6]. Subsequently, the reintegration protocol for NASA Langley's SPIDER (another fault-tolerant bus) was verified [11]. Recently, the authors verified two prominent physical-layer communication protocols: the Biphase Mark protocol (BMP), which is used in CD-player decoders, Ethernet, and Tokenring, and the 8N1 protocol, which is used in universal asynchronous receiver-transmitters (UARTs) [2]. The verification of the 8N1 protocol uncovered a significant error in the temporal constraints in an industrial technical note.

To motivate why infinite-bmc induction is of interest in real-time verification, consider that our verification of BMP in [2] resulted in an orders-of-magnitude reduction in effort as compared to the protocol's previous formal verifications using mechanical theorem proving. Our verification required 5 invariants, whereas a published proof using the mechanical theorem prover PVS required 37 [14]. Using infinite-bmc induction, proofs of the 5 invariants were completely automated, whereas the PVS proof initially required some 4000 user-supplied proof directives in total. Another proof using PVS is so large that the tool required 5 hours just to *check* the manually-generated proof whereas the SAL proof is generated automatically in seconds [9]. BMP has also been verified by J. Moore using Nqthm, a precursor to ACL2, requiring him to develop a model of asynchronous communication in the language of Applicative Common Lisp (Moore cites this as being one of the "best ideas" of his career, and at the time, it was a significant advancement in the formal analysis of physical-layer protocols) [10].[2]

Partially-parameterized verifications of BMP have also been done using the Uppaal and Hytech real-time model checkers [7, 8]. SAL is not specifically a real-time model checker; time is modeled using the real numbers, and real-time constraints are captured as formulas (usually linear inequalities) over the reals. Real-time model checkers are automated; in infinite-bmc induction provers, whereas each proof attempt is automated, the user iteratively builds up invariants (by refinement from automatically-generated counterexamples on failed proof attempts) to ensure a proof attempt succeeds. With respect to expressiveness, the theory that real-time model checkers decide is weaker than the theory decided by contemporary SMT solvers. For example, SRI's Yices is a SMT solver for the satisfiability of (possibly quantified) formulas containing uninterpreted functions, real and integer linear arithmetic, arrays, fixed-size bit-vectors, recursive datatypes, tuples, records, and lambda expressions [5].

Finally, we take a moment to delineate the contribution of this paper beyond the authors' previous work in physical-layer protocol verification [2]. There, verifications of real-time *implementations* of the 8N1 protocol and BMP were completed. In this paper, we demonstrate that the same proof technique can be used to complete a refinement proof, and that refinement allows for finite-state model composition. In both papers, we prove the same safety property holds, albeit in this paper, the property is over a finite-state model, so its formulation is simpler than its real-time equivalent from the authors' previous work. Additionally, we present improvements to the protocol model including (1) a more efficient model of metastability: rather than having additional state variables representing metastable states, the transition relation is made less deterministic, and (2) the effects of the environment on the protocol's behavior are decomposed into a separate environmental constraint module.

2.2 The 8N1 Protocol

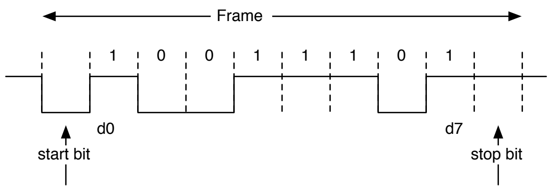

Figure 1. 8N1 Data Transmission

The 8N1 protocol is a physical-layer protocol commonly used in UARTs for serial communication between two independently-clocked and unsynchronized hardware units. Figure 1 illustrates the 8N1 encoding scheme implemented by the sender's UART. A *frame* is a sequence of bits that includes a start bit, eight data bits, and a stop bit. Data bits are encoded by the identity function – a 1 is a 1 and a 0 is a 0. The receiver must estimate the transmitter's clock based

[2]See http://www.cs.utexas.edu/users/moore/best-ideas/.

upon the transitions in the data stream which are guaranteed to occur only once per frame. The central design issue for a *data decoder* is reliably extracting a clock signal from the data stream. Once the location of the clock events is known, extracting the data is relatively simple. Although the clock events have a known relationship to signal transitions, detecting these transitions precisely is usually impossible because of distortion in the signal around the transitions due to the transmission medium, clock jitter, and other effects. A fundamental assumption is that the transmitter and receiver of the data do not share a common time base and hence the estimation of clock events is affected by differences in the reference clocks used. Constant delay is largely irrelevant; however, transition time and variable delay (e.g., jitter) are not. Furthermore, differences in receiver and transmitter clock phase and frequency are significant. A correctness proof of an 8N1 decoder must be valid over a range of parameters defining limits on jitter, transition time, frequency, and clock phase.

2.3 The Symbolic Analysis Laboratory (SAL)

SAL has a high-level modeling language for specifying transition systems. A transition system is specified by a *module*. A module consists of a set of state variables (declared to be *input*, *output*, or *local*) and guarded transitions. A transition is *enabled* if its guard is true. Of the enabled transitions in a state, one is nondeterministically executed. When a transition is exercised, the next-state values are assigned to variables; for example, in the guarded transition, `G --> a' = a - 1; b' = a`, if the guard `G` holds and the transition is exercised, then in the next state, the variable `a` is decremented by 1 and the variable `b` is updated to the previous value of `a`. In the language of SAL, "`;`" denotes statement separation, not sequential composition (thus, variable assignments can be written in any order); furthermore, in a variable assignment, next-state variable values of other variables can be referenced. If no variables are updated in a transition (i.e., `G -->`), the state idles.

Modules can be composed both synchronously (`||`) and asynchronously (`[]`), and composed modules communicate via shared variables. In a synchronous composition, a transition from each module is simultaneously applied; a synchronous composition is deadlocked if either module has no enabled transition. In an asynchronous composition, an enabled transition from exactly one of the modules is nondeterministically applied.

The language is typed, and predicate sub-types can be declared. Types can be both interpreted and uninterpreted, and base types include the reals, naturals, and booleans; array types, inductive data-types, and tuple types can be defined. Both interpreted and uninterpreted constants and functions can be specified. Parameterized values are represented as uninterpreted constants from some parameterized type.

Bounded model checkers are usually used to find counterexamples, but they can also be used to prove invariants by induction over the state space [3]. SAL supports k-*induction*, a generalization of the induction principle, which can prove some invariants that may not be strictly inductive. Let $(S, I, \rightarrow)$ be a transition system where S is a set of states, $I \subseteq S$ is a set of initial states, and $\rightarrow$ is a binary transition relation. If k is a natural number, then a k-*trajectory* is a sequence of states $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ (a 0-trajectory is a single state). Let k be a natural number, and let P be property. The k-induction principle is then defined as follows:

- *Base Case*: Show that for each k-trajectory $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ such that $s_0 \in I$, $P(s_j)$ holds, for $0 \leq j < k$.
- *Induction Step*: Show that for all k-trajectories $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$, if $P(s_j)$ holds for $0 \leq j < k$, then $P(s_k)$ holds.

The principle is equivalent to the usual transition-system induction principle when $k = 1$. In SAL, the user specifies the depth at which to attempt an induction proof, but the attempt itself is automated. The main mode of user-guidance in the proof process we use is in iteratively building up *disjunctive invariants* [12] by strengthening a conjectured invariant based on the counterexamples returned by SAL in a failed proof attempt [2]. The conjectured invariant is strengthened by adding additional disjuncts, based on the returned counterexamples, representing a configuration of the system not covered by the original invariant. Disjunctive invariants contrast to the traditional approach of strengthening an invariant by adding a *conjunct*. The benefit of disjunctive invariants is that each disjunct need only cover a "special case" of the system, while each conjunct needs to hold in every configuration of the system's state. We describe in more detail its application to the refinement proof in Section 5.2.

By incorporating a SMT solver, SAL can do k-induction proofs over infinite-state transition systems. Finally, SAL also has finite-state BDD-based and bounded model checkers as well as other tools such as a finite-state deadlock checker and simulator.

3 A Finite-State Specification of the 8N1 Protocol

We begin by presenting a generic cross clock-domain protocol model. Its components are illustrated in Figure 2. There is a transmitter (`tx`) and a receiver (`rx`), each of

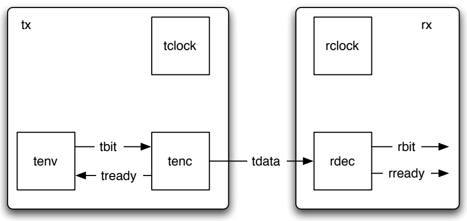

Figure 2. System Block Diagram

which contains subcomponents. The transmitter contains an encoder (`tenc`) that encodes data according to the protocol, an environment (`tenv`) that supplies data, and a local clock (`tclock`). The receiver contains a decoder (`rdec`) that decodes the data according to the protocol and a local clock (`rclock`).

Each block is modeled as a module in SAL, and the transmitter's modules are synchronously composed with each other, as are the receiver's modules. The entire system is the asynchronous composition of the transmitter and receiver:

```
tx : MODULE = tclock || tenv || tenc;
rx : MODULE = rclock || rdec;
system : MODULE = tx [] rx;
```

The modules `tclock` and `rclock` are instantiated for the 8N1 protocol in Section 3.1, and the modules `tenv`, `tenc`, and `rdec` are instantiated in Section 3.2. The verification of the finite-state 8N1 specification is presented in Section 3.3.

3.1 The Clocks

Following Figure 1, the transmitter and receiver specifications each have 10 states, labeled 0 – 9, corresponding to the phases of data transmission in a frame. The transmitter is idle (repeatedly transmitting the stop bit) in state 9, transmits the start bit in state 0, and transmits data bits in states 1-8. Likewise, the receiver begins in state 9 awaiting a start bit, decodes data in states 1-8, and scans for the stop bit in state 0. An execution is therefore correct if it has the following sequence of states, where "…" denotes indefinite idling and "$\cdots$" denotes the removal of a finite number of states in the sequence for readability:

$$
\begin{aligned}
&(\texttt{tstate}, \texttt{rstate}) = \\
&\quad (9,9),(9,9),\ldots,(0,9),\ldots,(0,0),(1,0),(1,1),\cdots, \\
&\quad (8,7),(8,8),(9,8),(9,9),\ldots
\end{aligned}
$$

Thus, the transmitter may transition between states whenever the states of the transmitter and receiver are equal. We capture this requirement with the `tclock` module in Figure 3. The receiver's clock in Figure 4 is similarly modeled, although it is allowed to transition only when the states of the transmitter and receiver are unequal or the transmitter has not yet sent a start bit. The two clocks serve to constrain the interleavings of the sender and receiver. The clocks constrain both in the finite-state and infinite-state models (see Section 4.1), and the refinement principally involves refining the clock modules, which determine the interleavings of the transmitter and receiver.

```
STATE : TYPE = [0..9];

tclock : MODULE =
BEGIN
  INPUT rstate : STATE
  INPUT tstate : STATE
  TRANSITION
    [ tstate = rstate --> ]
END;
```

Figure 3. Transmitter Clock (Finite-State)

```
rclock : MODULE  =
BEGIN
  INPUT tstate : STATE
  INPUT rstate : STATE
  TRANSITION
    [    rstate /= tstate
      OR tstate = 9 --> ]
END;
```

Figure 4. Receiver Clock (Finite-State)

3.2 The Environment, Encoder and Decoder

```
tenv : MODULE =
BEGIN
  INPUT   tready : BOOLEAN
  OUTPUT  tbit   : BOOLEAN
  TRANSITION
  [
      tready --> tbit' IN {TRUE, FALSE}
   [] ELSE   -->
  ]
END;
```

Figure 5. Transmitter Environment (Finite-State)

The transmitter's environment `tenv` (Figure 5) represents an arbitrary data source. The module has a single input `tready` and a single output `tbit` (see Section 6 for a description of a model that has shift registers producing data). Whenever `tready` is true, the environment generates a new (random) output.

```
tenc : MODULE =
BEGIN
  OUTPUT  tdata  : BOOLEAN
  OUTPUT  tstate : STATE
  OUTPUT  tready : BOOLEAN
  INPUT   tbit   : BOOLEAN
  INITIALIZATION
    tstate = 9;
  DEFINITION
    tready = state /= 8;
    tdata = (tstate = 9) OR tbit;
  TRANSITION
  [
      tstate = 9 -->
        tstate' = IF tbit'
                   THEN 9 ELSE 0
                   ENDIF;
   [] tstate < 9 --> tstate' = tstate + 1;
  ]
END;
```

Figure 6. Transmitter Encoder (Finite-State)

The transmitter's encoder `tenc` (Figure 6) has one input variable, `tbit`, which is the output of `tenv`. The encoder has three output variables: `tdata` is the data transmitted to the receiver, `tstate` is the transmitter's state, and `tready` is the encoder's signal to `tenv` whenever a new bit is required from the environment. The environment (`tenv`) decides when a new frame of data is to be transmitted; this decision is implemented by providing an initial start bit (`FALSE`). The encoder module has two guarded transitions. The first guarded transition defines the behavior in the idle state (9), which is exited when the environment produces a start bit. The second transition simply advances the transmitter's state; the ready signal is suppressed in state 8 to indicate to the environment that a stop bit (`TRUE`) will be inserted in the data stream.

The receiver's decoder `rdec` (Figure 7) has a design similar to `tenc`. Of its three variables, `tdata` is the data sent by the transmitter (i.e., the output variable of `tenc`), `rstate` is the receiver's state, and `rbit` records the value received from the transmitter. The module has three transitions: the first allows the receiver to idle waiting for the start bit, the second transitions to the "start state", and the last transition governs states in which it is sampling for data. The data is ready to be consumed when the receiver is in one of states 1-8.

```
rdec : MODULE =
BEGIN
  INPUT  tdata  : BOOLEAN
  OUTPUT rstate : STATE
  OUTPUT rbit   : BOOLEAN
  INITIALIZATION
    rbit  = TRUE;
    rstate = 9;
  TRANSITION
    [   rstate  = 9 -->
     [] rstate  = 9 AND NOT tdata -->
          rbit' = FALSE;
          rstate' = 0;
     [] rstate /= 9 -->
          rbit' = tdata;
          rstate' = rstate + 1;
]
END;
```

Figure 7. Receiver Decoder (Finite-State)

The transitions of `rdec` are under-constrained – even when the transmitter provides a start bit, the receiver may continue to stutter in state 9. Stuttering is an essential characteristic of physical reality where changes to signals take time to propagate and to be correctly sampled. In the design of the implementation, we introduce a constraint process that limits this stuttering based upon real-time properties of the local clocks and which guarantees that the stuttering is finite.

3.3 Correctness

Informally, the protocol is correct if the output of the decoder is the same as the input of the encoder whenever the transmitter and receiver are in the same state (i.e., the receiver has just received the message sent), and the receiver is neither in state 0 or 9 – recall that in state 9, the start bit is being sent, and the model captures the possibility that the receiver misses the bit sent by the transmitter due to their clocks being unsynchronized. The theorem is captured by the following LTL specification:

```
Thm: THEOREM system |-
  G(tstate = rstate AND rstate /= 9 =>
      rbit = tbit);
```

The theorem is proved using SAL's BDD-based model checker. We also use the BDD-based model checker to demonstrate that the possible sequence of states of `system` is exactly characterized by the sequence of states listed in Section 3.1. As a convenience, we verify this through three small theorems.

```
StateThm1 :  THEOREM system |-
```

```
  G(   rstate = tstate
    OR ((rstate + 1) MOD 10) = tstate);

StateThm2 :  THEOREM system |-
 G(FORALL (i : STATE) :
   (i = rstate AND i = tstate) =>
     X(   (    i = rstate
          AND (i + 1) MOD 10 = tstate)
       OR (    i = rstate
          AND i = tstate AND i = 9)));

StateThm3 : THEOREM system |-
  G(FORALL (i : STATE) :
    (    i = rstate
     AND (i + 1) MOD 10 = tstate) =>
       X(   (    (i + 1) MOD 10 = rstate
            AND (i + 1) MOD 10 = tstate)
         OR (    i = 9 AND i = rstate
            AND 0 = tstate)));
```

Finally, SAL's deadlock checker ensures liveness of the model; given the assumption that the receiver detects a start bit, the system will eventually return to an "idle" state.

```
Liveness: THEOREM system |-
   G(F(rstate /= 9) =>
       F(tstate = 9 AND rstate = 9));
```

4 8N1 Implementation

An implementation of the 8N1 protocol utilizes local clocks at the transmitter and receiver to achieve the synchronization necessary to reliably transmit data. These clocks are only loosely synchronized – each has a nominal frequency, but there may be both constant and dynamic frequency errors. Furthermore, the electrical properties of data transmission lead to both transmission delay and sampling issues due to "reading" a changing signal. The model presented in this section captures these various properties.

The overall structure of the implementation is similar to the model presented in Section 3; the essential difference is that a constraint module is introduced to capture the real-time manifestations of asynchrony between the clocks (Section 4.2).

```
tx_rt : MODULE = tenv || tenc || tclock_rt;
rx_rt : MODULE = rdec_rt || rclock_rt;
system_rt : MODULE =
  (rx_rt [] tx_rt) || constraint;
```

4.1 Clocks

The clocks are modeled as *timeout automata* in which the progress of time is enforced cooperatively by (possibly nondeterministically) updating variables called *timeouts* over the real numbers [6]. In the 8N1 protocol, the receiver and transmitter have the timeouts `rclk` and `tclk` that mark the real-time at which they will respectively make transitions. Each respective module representing the receiver and transmitter is allowed to execute only if its timeout equals the value of `time(rclk, tclk)`, which is defined to be the minimum of the timeout variables:

```
TIME : TYPE = REAL;
time(t1 : TIME, t2: TIME): TIME =
  IF t1 <= t2 THEN t1 ELSE t2 ENDIF;
```

```
TPERIOD : {x : TIME | 0 < x};
TSETTLE : {x : TIME | 0 <= x
            AND x < TPERIOD};
TSTABLE : TIME =
  TPERIOD - TSETTLE;

tclock_rt : MODULE =
BEGIN
  INPUT  rclk   : TIME
  OUTPUT tclk   : TIME
  INITIALIZATION
    tclk  IN
      {x : TIME | 0 <= x AND
         x <= TSTABLE};
  TRANSITION
  [
    tclk = time(tclk, rclk) -->
      tclk' = tclk + TPERIOD;
  ]
END;
```

Figure 8. Transmitter Clock (Infinite-State)

The transmitter clock `tclock_rt` (Figure 8) executes at a fixed, but arbitrary rate – without loss of generality, all of the relative frequency errors are captured in the receiver clock relative to the transmitter frequency. The transmitter clock period consists of a settling phase (`TSETTLE`) and a stable phase (`TSTABLE`). The settling phase captures both propagation delay as well as setup requirements at the receiver (to be discussed in Section 4.2). `TSETTLE` and `TSTABLE` are uninterpreted constants; however, they are parameterized, which allows us to verify the model for any combination of settling time and receiver clock error. The transmitter settling time can be used to capture the effects of jitter and dispersion in data transmission as well as jitter in the transmitter's clock. In the case of the settling period, the model can be viewed as less deterministic than an actual implementation which might reach stable transmission values sooner.

The receiver clock `rclock_rt` (Figure 9) is more complicated than the transmitter clock because of the manner in which a UART is implemented. Consider Figure 1.

```
timeout(min : TIME, max : TIME) :
  [TIME -> BOOLEAN] =
    {x : TIME | min <= x AND x <=  max};

rclock_rt : MODULE  =
BEGIN
  INPUT tclk   : TIME
  INPUT rstate : STATE
  OUTPUT rclk  : TIME
  INITIALIZATION
    rclk IN
      { x : TIME | 0 <= x AND
                   x < RSCANMAX };
  TRANSITION
  [
    rclk = time(rclk, tclk) --> rclk' IN
      IF (rstate' = 9)
      THEN timeout(rclk + RSCANMIN,
                   rclk + RSCANMAX)
      ELSIF (rstate' = 0)
      THEN timeout(rclk + RSTARTMIN,
                   rclk + RSTARTMAX)
      ELSE timeout(rclk + RPERIODMIN,
                   rclk + RPERIODMAX)
      ENDIF;
  ]
END;
```

Figure 9. Receiver Clock (Infinite-State)

There may be an arbitrary "idle" period between frames during which the signal is high (TRUE). The behavior of a UART receiver is to "scan" for the high-to-low transition that marks the beginning of a frame. Once this transition is detected, the receiver predicts, based upon its local time reference, the middle of the 8 data and 1 stop bit times. There are two different intervals used for this prediction – the time between the detected "start" transition and the middle of the first data bit and the "period" between successive data samples. In an implementation, the bit period is generally an integer multiple of the scan time and the start interval is 1.5 times the bit period. In most implementations, the bit time of the receiver is approximately that of the transmitter; however, in practice jitter and frequency errors mean that each measurement interval is subject to error. Again we associate all errors with the receiver and assume that the transmitter runs at a constant rate.

The various receiver clock periods are expressed in terms of linear constraints that define lower and upper bounds for "SCAN", "START", and "PERIOD" (Figure 10). The constraints give the lower and upper bounds for the receiver's timeout variable to be updated. Determining the necessary constraints is achieved by assuming the worst-case error (minimum or maximum) and then determining how temporal errors accumulate by the 10th bit time (the stop bit). Informally, the correct behavior of the protocol requires that all signal samples other than the initial scan fall during the "stable" portion of the transmitter clock [2].

```
RSCANMIN : {x : TIME | 0 < x};
RSCANMAX : {x : TIME | RSCANMIN <= x AND
                        x < TSTABLE};

RSTARTMIN :
  {x : TIME | TPERIOD + TSETTLE < x};
RSTARTMAX : {x : TIME | RSTARTMIN <= x AND
                      x < 2 * TPERIOD -
                      TSETTLE - RSCANMAX};

RPERIODMIN :
  {x : TIME | 9 * TPERIOD + TSETTLE <
              RSTARTMIN + 8 * x};
RPERIODMAX :
  {x : TIME | RPERIODMIN <= x AND
                TSETTLE + RSCANMAX +
                RSTARTMAX + 8 * x <
                  10 * TPERIOD};
```

Figure 10. Real-Time Linear Constraints

4.2 Modeling Metastability

A fundamental issue for the implementation is that it is not possible to reliably sample an asynchronous signal unless that signal is guaranteed to be stable for a period around the sampling point. Where the physical implementation is realized with a flipflop, changes occurring too soon before the sampling point are said to violate the "setup time" requirement of the flip-flop while changes occurring too soon after the settling point are said to violate the "hold time" requirement. Either violation may cause the flip-flop to enter a *metastable* state in which the input is neither "one" nor "zero" and which may persist indefinitely. A constraint model captures the effects of metastability.

Again, without loss of generality, we consider only setup time violations. We can incorporate the hold-time requirement into the setup-time requirement simply by changing our perspective of where the sampling event occurs relative to the clocks. To model the possibility of metastability (and hence random sampling of signals), we use a decoder `rdec_rt` (Figure 11) in which all input sampling results in the selection of a random value. To constrain the nondeterminism of the decoder, we introduce a constraint module (Figure 12) that monitors changes on the input signal `tdata` (when `tclk' /= tclk`) and forces the decoder to choose the correct value whenever the setup time requirements are met.

The constraint module requires that `rdec_rt` select the "correct" value (i.e. `tdata`) whenever the input signal has met the stability requirements. These stability requirements

```
rdec_rt : MODULE =
BEGIN
  OUTPUT rstate : STATE
  OUTPUT rbit   : BOOLEAN
  INITIALIZATION
    rbit  = TRUE;
    rstate = 9;
  TRANSITION
    [
        rstate  = 9 --> rbit' = TRUE;
     [] rstate  = 9 --> rbit' = FALSE;
                        rstate' = 0;
     [] rstate /= 9 -->
          rbit' = {TRUE,FALSE;}
          rstate' = rstate + 1;
    ]
END;
```

Figure 11. Receiver Decoder (Infinite-State)

```
constraint : MODULE =
BEGIN
  INPUT  tclk     : TIME
  INPUT  rclk     : TIME
  INPUT  rbit     : BOOLEAN
  INPUT  tdata    : BOOLEAN
  LOCAL  stable   : BOOLEAN
  LOCAL  changing : BOOLEAN
  DEFINITION
    stable = NOT changing
             OR tclk - rclk < TSTABLE;
  INITIALIZATION
    changing = FALSE
  TRANSITION
  [
      rclk' /= rclk AND
      (stable => rbit' = tdata) -->
   [] tclk' /= tclk -->
        changing' = (tdata' /= tdata)
  ]
END;
```

Figure 12. Constraint Module (Infinite-State)

depend upon when tdata last changed its value. The timing constraints defined in Figure 10 force the settling time to be less that TPERIOD.

5 Proving Refinement

With both a finite-state specification and an infinite-state implementation in hand, we can now prove the latter refines the former. More specifically, every possible interleaving of the transmitter and receiver in the implementation is a possible interleaving of the specification. Paraphrasing Abadi and Lamport, **I** implements **S** if every externally visible behavior of **I** is also allowed by **S** [1]. To prove that **I** implements **S** it is sufficient to prove that if **I** allows the behavior

$$\langle\langle(e_0, z_0), (e_1, z_1), (e_2, z_2), ...\rangle\rangle$$

where each e_i is an externally-visible state, and where each z_i is an internal state (the remainder of the state), then there exist internal states y_i such that **S** allows

$$\langle\langle(e_0, y_0), (e_1, y_1), (e_2, y_2), ...\rangle\rangle$$

5.1 Guard Weakening in SAL

In general, an Abadi-Lamport style refinement proof can be difficult and may rely upon the introduction of history and prophecy variables. For the class of protocols we consider in this paper, the refinement mappings are straightforward. The basic transformation rule that we apply is *guard weakening* over SAL's guards to show that the guards of the implementation imply the guards of the specification. Consider a module consisting of the following set of guarded transitions:

$$G_0 \to S_0[] \dots []G_N \to S_N$$

where each G_i is a predicate over the state variables, and each S_i is a set of possibly nondeterministic next-state variable assignments. If G_i implies G'_i for each i, then the following set of guarded commands allow a superset of behaviors and are therefore are a specification for the former:

$$G'_0 \to S_0[] \dots []G'_N \to S_N$$

Theorems of the form $G_i \Rightarrow G'_i$ are the *refinement conditions*. The semantics of synchronous and asynchronous composition in SAL are such that the effect of weakening any guard is to expand the set of legal behaviors [3]; we have covered the case of asynchronous composition above. In the case of synchronous composition, observe that the following equivalence holds:

$$(G_0 \to S_0 \parallel G_1 \to S_1) \equiv (G_0 \wedge G_1 \to S_0; S_1)$$

(Recall from Section 2.3 that ; is commutative.) Weakening either G_0 or G_1 has the effect of weakening $G_0 \wedge G_1$. Indeed, by weakening either guard to True, its associated module can be eliminated (effectively abstracting away the module's outputs). The key observation is that an unconstrained input is treated by SAL as having any possible value – eliminating a module and its associated outputs increases the nondeterminism of the corresponding inputs. Thus, in a synchronous composition P || Q, guard weakening can be generalized to module elimination: P and Q are both specifications of P || Q. A corollary to the equivalence above allows us to "split" a process:

$$(G_0 \to S_0) \equiv ((G_0 \to skip) \parallel (TRUE \to S_0))$$

5.2 Temporal Refinement for the 8N1 Protocol

In this section, we describe the refinement conditions proved in SAL for the 8N1 protocol. In short, the refinement is principally over the clock modules of the finite-state and infinite-state models. Recall that `system_rt` is the composed module `(rx_rt [] tx_rt) || constraint`. From the discussion of module elimination above, the `constraint` module can be eliminated, and the composition `rx_rt [] tx_rt` is a specification of `system_rt`. For the transmitter implementation module `tx_rt`, we prove the single guard in `tclock_rt` (Figure 8) implies that the single guard in `tclock` (Figure 3) holds:

```
tclock_thm : THEOREM system_rt |-
  G(tclk = time(tclk, rclk) =>
      tstate = rstate);
```

The proof is automated in SAL using infinite-bmc induction. The proof requires a small invariant bounding the maximum skew between the clocks as a function of the transmitter's and receiver's respective states.

This is the only refinement condition required for `tx_rt`: the other two modules, `tenv` and `tenc`, are identical in the specification and implementation. As for the module `rx_rt`, recall that the module is the parallel composition `rdec_rt || rclock_rt`. The refinement condition and proof for `rclock_rt` (Figure 9) is exactly analogous to that for `tclock_rt`.

This leaves only `rdec_rt` to refine. To complete the refinement proof, we prove a refinement condition that slightly generalizes guard weakening. The module `rdec_rt` (Figure 11) is actually less deterministic than is `rdec` (Figure 7), but its nondeterminism is limited by the constraint module. In simple guard weakening, the next-state variable assignments in the specification and implementation are identical, but for the modules `rdec` and `rdec_rt`, they are not. In this case, the hypothesis of the refinement condition is a conjunction of the implementation's guard and the next-state variable values produced by the implementation's transition. Likewise, the antecedent of the refinement condition is a conjunction of the specifications guard and the next-state variable values produced by the specification's corresponding transition.

Applying this reasoning to `rdec_rt` and `rdec`, we prove three theorems, one for each of the three corresponding guarded transitions in the two modules:

```
rdec_thm1 : THEOREM system_rt |-
  G(rstate = 9 => rbit);

rdec_thm2 : THEOREM system_rt |-
  G(    rstate = 9 AND X(NOT rbit)
    AND X(rstate = 0) => NOT tdata);

rdec_thm3 : THEOREM system_rt |-
  G(FORALL (i : [0..8]) :
      rstate = i AND X(rstate = i+1) =>
        X(rbit) = tdata);
```

Each of these theorems is proved using a disjunctive invariant lemma [12, 2]. For the 8N1 protocol, the disjunctive invariant covers the two special-case configurations – when the transmitter and receiver are idle and when the transmitter has sent a start bit that is not detected – as well as the two general-case configurations – when `tstate = rstate` and when `tstate = rstate + 1`.

6 Discussion

We have presented by example how to use a model checker and SMT solver to prove temporal refinement of the 8N1 protocol. We have also applied this method to prove refinement of the Biphase Mark protocol, and those specifications and proofs are also available.[1] These results should be extensible to other physical-layer protocols. The purpose of this paper is not to delineate the class of real-time specifications to which this approach is feasible but rather to demonstrate that it is possible. The theoretical considerations of Abadi-Lamport style refinement for real-time systems (particularly with linear constraints) deserve additional study.

As mentioned in Section 1, one motivation for these refinement results comes from the desire for compositional reasoning. We have completed a infinite-state model of the 8N1 protocol composed with shift registers (with widths of eight bits) for the sender and receiver, respectively.[1] One would like to prove that after a round of execution of the 8N1 protocol, the receiver's shift register contains the value that was in the sender's shift register before the round. Doing the proof directly in an infinite-state model requires augmenting an invariant with properties of the shift registers. However, the proof is automatic using BDDs in a finite-state model of the system augmented with shift registers. Then, only the cross-clock domain protocol needs to be refined to a real-time implementation – the shift registers specification is the same in both the specification and implementation.

As pointed out by an anonymous reviewer, although we present an methodology for temporal refinement, the real-time implementation can itself be refined using data refinement techniques. The data-refined implementation should still inherent the safety properties from the untimed, data-abstract specifications. Carrying out data refinement on our model is future work.

While this paper demonstrates the feasibility of automating parts of refinement proofs utilizing SAL, the work was somewhat hampered by the lack of a well-developed algebraic theory for the SAL language and by the need to apply algebraic reasoning by hand. Automated model checking techniques have matured sufficiently to prove the tedious refinement conditions presented in this paper, but combining SAL with a system capable of algebraic reasoning (e.g., an interactive theorem prover) would both increase the assurance and decrease the tedium of deriving the implementation and refinement conditions from the specification.

Another anonymous noted that refinement does not preserve liveness properties. We have proved liveness for the finite state model but not for the implementation. There is a possibility that the implementation is vacuous. Because the implementation is infinite-state, we cannot directly prove LTL liveness properties about it. As an approximation, we can attempt to prove some safety property of the implementation that should be false (e.g., `G(tstate > 0)`. If the system is deadlocked, then the property may be proved. If the property is not proved, then either the system is not deadlocked, or the property is not k-inductive, for the value of k and lemmas used in the proof effort. We call the these safety properties "poor man's liveness properties".

Finally, as this work is primarily a case-study, we provide some "lessons learned" to inform similar future endeavors. In Section 2, we gave some metrics comparing our verification efforts to previous work. In our work, the vast majority of our time was spent generalizing the model. Developing the "right" modeling abstractions took a few weeks of effort. Instantiating the general model with another physical-layer protocol would probably take a day of effort (the effort required is speculative since we developed the general model by abstracting the specific protocols). Although we conjecture that carrying out the proof via SMT and model checking is at least an order-of-magnitude faster than using mechanical theorem proving, it is still the most onerous aspect of the verification. The particular difficulty results from building up the main disjunctive invariant for a sufficiently small value of k. Developing the invariant for a new protocol may take on the order of a few days of effort.

Acknowledgments

We thank our anonymous reviewers for their suggestions and corrections.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] G. M. Brown and L. Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, pages 58–72, 2006. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.

[3] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Computer-Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[4] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Computer-Aided Verification, CAV'03*, volume 2725 of *LNCS*, 2003.

[5] B. Dutertre and L. de Moura. Yices: an SMT solver. Available at `http://yices.csl.sri.com/`, August 2006.

[6] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *FORMATS/FTRTFT*, pages 199–214, 2004.

[7] T. Henzinger, J. Preussig, and H. Wong-Toi. Some lessons from the Hytech experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892, 2001.

[8] T. S. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. Technical Report RS-01-5, BRICS, University of Aarhus, January 2001.

[9] D. V. Hung. Modelling and verification of biphase mark protocols using PVS. In *Proceedings of the International Conference on Applications of Concurrency to System Design (CSD'98),* Aizu-wakamatsu, Fukushima, Japan, March 1998, pages 88–98. IEEE Computer Society Press, 1998.

[10] J. S. Moore. A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. *Formal Aspects of Computing*, 6(1):60–91, 1994.

[11] L. Pike and S. D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 286–289, New York, NY, USA, 2005. ACM Press.

[12] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In *Computer-Aided Verification, CAV'00*, volume 1855 of *LNCS*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag.

[13] J. Rushby. Harnessing disruptive innovation in formal verification. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2006. Available at `http://www.csl.sri.com/users/rushby/abstracts/sefm06`.

[14] F. W. Vaandrager and A. L. de Groot. Analysis of a Biphase Mark Protocol with Uppaal and PVS. Technical Report NIII-R0455, Nijmegen Institute for Computing and Information Science, 2004.